Comptage de solutions en exploitant la structure du graphe de contraintes

Aurélie Favier, Simon de Givry, Philippe Jégou

▶ **To cite this version:**

Aurélie Favier, Simon de Givry, Philippe Jégou. Comptage de solutions en exploitant la structure du graphe de contraintes. Cinquièmes Journées Francophones de Programmation par Contraintes, Orléans, juin 2009, Jun 2009, Orléans, France. pp.25-35. hal-00387846

HAL Id: hal-00387846
https://hal.science/hal-00387846

Submitted on 25 May 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Comptage de solutions en exploitant la structure du graphe de contraintes

Aurélie Favier, Simon de Givry et Philippe Jégou

INRA MIA Toulouse, France
Université Paul Cézanne, Marseille, France
{afavier,degivry}@toulouse.inra.fr, philippe.jegou@univ-cezanne.fr

Abstract

Ce papier traite du problème de comptage du nombre de solutions d'un CSP, dénoté #CSP. Ce problème très difficile a de nombreuses applications en informatique, en particulier en Intelligence Artificielle, et aussi en physique statistique. Des progrès récents ont été fait par des méthodes de recherche, telle que BTD [16], qui exploitent la structure du graphe de contraintes dans le but de résoudre des CSPs efficacement. Nous proposons d'adapter BTD pour résoudre le problème #CSP. La méthode de comptage exacte résultante a dans le pire des cas une complexité temporelle exponentielle en un paramètre graphique spécifique appelé *largeur d'arbre*. Pour des problèmes ayant un graphe peu dense mais une grande largeur d'arbre, nous proposons une méthode itérative qui approche le nombre de solutions en résolvant une partition de l'ensemble des contraintes en une collection de sous-graphes partiels triangulés. Sa complexité temporelle est exponentielle en la taille de la plus grande clique (aussi appelé *clique number*) du problème d'origine, taille qui peut être beaucoup plus petite que la largeur d'arbre. Des expérimentations sur des problèmes aléatoires CSP structurés et des benchmarks SAT montrent l'efficacité pratique de nos approches.

1 Introduction

Le formalisme des problèmes de satisfaction de contraintes (CSP) offre un cadre général pour représenter et résoudre de nombreux problèmes. Déterminer si une solution existe est un problème NP-complet. Un problème plus difficile encore consiste à compter le nombre de solutions. Ce problème noté #CSP est connu pour être #P-complet [26]. Ce problème a de nombreuses applications en informatique, en particulier en IA, par exemple en raisonnement approché [23], en diagnostique [19], en révision de croyance [5], comme heuristique pour guider la recherche d'une solution dans les CSPs [17], ainsi que dans d'autres domaines en dehors de l'informatique, tels que la physique statistique [3] ou en biochimie pour la prédiction de structures de protéines [20].

Dans la littérature, deux principales approches ont été étudiées. D'un côté, les méthodes calculant le nombre exact de solutions et de l'autre côté des méthodes approchées. Pour les méthodes exactes, l'approche naturelle est d'étendre les méthodes systématiques telles que FC [15] ou MAC [24] dans le but d'énumérer toutes les solutions. La complexité est bornée par $O(m.d^n)$ avec n le nombre de variables, m le nombre de contraintes, et d la taille maximum des domaines. Il est évident qu'avec cette approche, plus il y a de solutions,

plus cela prend du temps pour les énumérer.

Dans ce papier, nous sommes intéressés par des méthodes de recherche arborescente qui exploitent la structure des problèmes, offrant de meilleures bornes sur la complexité temporelle et spatiale. C'est le cas du compilateur de formules en logique propositionnelle en d-DNNF [6] et de la recherche dans des graphes ET/OU [8, 9] qui font tous les deux du comptage de solutions.

Nous proposons d'adapter l'algorithme Backtracking with Tree-Decomposition (BTD) [16] à #CSP. BTD avait été initialement proposé pour résoudre des CSPs structurés. Notre modification sur BTD est similaire à celle effectuée dans le cadre de la recherche ET/OU [8, 9]. Cependant BTD utilise la notion d'arbre de décomposition de clusters au lieu d'un pseudo-arbre, ce qui conduit naturellement BTD à utiliser un ordre dynamique de choix de variables à l'intérieur des clusters, tandis que la recherche ET/OU utilise plutôt un ordre statique.

La plus part du travail réalisé sur le comptage a été fait sur #SAT, le problème de comptage de modèles associé à SAT [26]. Les méthodes exactes pour #SAT étendent les résolveurs SAT systématiques, en ajoutant une analyse des composants [2] et de la mémorisation [25] pour améliorer les performances.

Les approches qui réalisent une approximation proposent une estimation du nombre de solutions. Elles proposent des algorithmes en temps polynômial ou exponentiel qui doivent fournir des approximations de qualité raisonnable avec des garanties théoriques sur la qualité de l'approximation ou non. De même que pour les méthodes exactes, les principaux travaux sur les méthodes d'approximation ont été fait sur #SAT. Ces approches effectuent soit un échantillonnage dans l'espace de recherche d'origine [27, 12, 10, 18], soit dans l'espace de recherche ET/OU [11]. Toutes ces méthodes, à l'exception de [27] procurent un minorant probabiliste du nombre de solutions avec un intervalle de confiance très resserré obtenu par des affectations aléatoires des variables jusqu'à trouver des solutions. Un inconvénient possible de ces approches est qu'elles peuvent ne trouver aucune solution dans le temps de calcul imparti à cause du fait qu'elles rencontrent uniquement des affectations partielles incohérentes. Pour des problèmes complexes de grande taille, cela conduit à des minorants du nombre de solutions nuls. Pour tenter de remédier à ce problème, il faut alors réaliser un réglage délicat des paramètres de ces méthodes, par exemple en changeant le nombre d'échantillons. Une autre approche consiste à réduire l'espace de recherche en ajoutant par exemple des contraintes XOR [13, 14]. Cependant, l'ajout de ces contraintes n'assure pas que le problème soit plus facile à résoudre.

Dans ce papier, nous proposons de relaxer le problème de comptage en effectuant un partitionnement des contraintes en une collection de sous-problèmes structurés triangulés. Chaque sous-problème est alors résolu en utilisant notre version modifiée de BTD. Cette tâche devrait être relativement facile si l'instance originale a un graphe peu dense[1]. Finalement, une approximation du nombre de solutions du problème complet est obtenue en combinant les résultats obtenus pour chaque sous-problème. La méthode d'approximation résultante, appelée `ApproxBTD` par la suite, donne aussi un majorant trivial du nombre de solutions. Les résultats expérimentaux montrent qu'une telle approche est intéressante pour sa rapidité et la qualité de son approximation.

D'autres méthodes de comptage fondées sur des relaxations ont été étudiées dans la littérature, telle que l'élimination de variables approchée et la propagation itérative dans un graphe de jointure [17], ou bien dans le contexte voisin de l'inférence Bayésienne, la propagation de croyance itérative et la méthode de suppression d'arêtes [4][2]. Ces approches ont le défaut de ne

[1] En fait, cela dépend de la largeur d'arbre des sous-problèmes, qui est bornée par la taille de la plus grande clique dans le problème complet. Dans le cas de graphes peu denses, nous nous attendons à ce que cette taille soit petite, ce qui est montré dans nos expérimentations. En pratique, notre méthode d'approximation ne sera efficace que si le CSP à résoudre ne contient pas de contraintes globales.

[2] Cette méthode commence par résoudre un sous-problème structuré en poly-arbre, par la suite augmenté par la restauration d'arêtes supprimées, jusqu'à au final résoudre le problème complet. A l'in-

pas exploiter la structure locale (aussi appelée *micro-structure*) des instances comme c'est le cas pour BTD, grâce à son emploi d'une cohérence locale et d'un ordre de choix de variables dynamique au sein des clusters.

Dans la section suivante, nous introduisons les notations et rappelons les principes de l'algorithme BTD. La section 3 décrit notre version modifiée de BTD. Dans la section 4, nous introduisons la méthode d'approximation ApproxBTD. Des résultats expérimentaux sur des CSPs aléatoires et des benchmarks SAT sont présentés dans la section 5. Enfin, nous donnons une conclusion dans la section 6.

2 Préliminaires

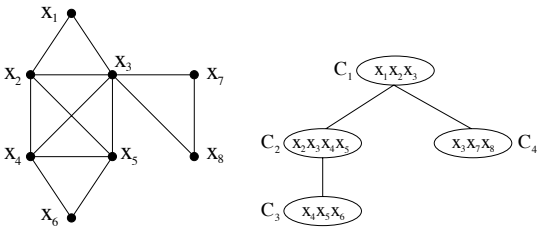
Un *problème de satisfaction de contraintes* (CSP) est défini par un tuple (X, D, C, R). X est un ensemble $\{x_1, \ldots, x_n\}$ de n variables. Chaque variable x_i prend ses valeurs dans le domaine fini d_{x_i} de D. Les variables sont soumises à un ensemble C de m contraintes. Chaque contrainte c est définie comme un ensemble $\{x_{c_1}, \ldots, x_{c_k}\}$ de variables. Une relation r_c (de R) est associée à chaque contrainte c telle que r_c représente l'ensemble des tuples autorisés sur $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Notons que nous pouvons également définir les contraintes par des fonctions usuelles ou des prédicats, par exemple. Etant donné $Y \subseteq X$ tel que $Y = \{x_1, \ldots, x_k\}$, une *affectation* des variables de Y est un tuple $\mathcal{A} = (v_1, \ldots, v_k)$ sur $d_{x_1} \times \cdots \times d_{x_k}$. Une contrainte c est *satisfaite* par $\mathcal{A}$ si $c \subseteq Y, (v_1, \ldots, v_k)[c] \in r_c$, elle est dite *violée* sinon. Nous écrirons l'affectation $(v_1, \ldots, v_k)$ sous la forme plus explicite $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$. La structure d'un CSP peut être représentée par le graphe (X, C), appelé le *graphe de contraintes*, dont les sommets sont les variables de l'ensemble X et il y a une arête entre deux sommets s'il existe une constrainte entre les variables correspondantes.

Dans [16] une nouvelle méthode est proposée pour résoudre les CSP. Cette méthode appelée BTD (pour Backtracking with Tree-Decomposition) est une méthode énumérative guidée par une décomposition arborescente du graphe de contraintes. Une *décomposition arborescente* [22] d'un graphe $G = (X, E)$ est une paire $(\mathcal{C}, \mathcal{T})$ avec $\mathcal{T} = (I, F)$ un arbre et $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ une famille de sous-ensemble de X, tels que chaque cluster $\mathcal{C}_i$ est un nœud de $\mathcal{T}$ et vérifie : (1) $\cup_{i \in I} \mathcal{C}_i = X$, (2) Pour chaque arête $\{x, y\} \in E$, il existe $i \in I$ avec $\{x, y\} \subseteq \mathcal{C}_i$, (3) Pour tout $i, j, k \in I$, si k est sur le chemin de i à j dans $\mathcal{T}$, alors $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$. La largeur d'arbre d'une décomposition arborescente $(\mathcal{C}, \mathcal{T})$ est égale à $\max_{i \in I} |\mathcal{C}_i| - 1$. La largeur d'arbre (tree-width) de G est la largeur minimale sur toutes les décompositions arborescentes de G. Notons que trouver une décomposition arborescente optimale est un problème NP-dur [1]. Cependant, nous pouvons facilement calculer un bonne décomposition arborescente en utilisant la notion de *graphes triangulés*. Une décomposition arborescente est calculée par triangulation (*i.e* ajout d'arêtes au) du graphe de contraintes pour qu'il devienne *triangulé*[3] afin d'y rechercher les cliques maximales dans le graphe de contrainte triangulé. La figure 1(b) présente une décomposition arborescente possible pour le graphe de la figure 1(a). Nous avons $\mathcal{C}_1 = \{x_1, x_2, x_3\}$, $\mathcal{C}_2 = \{x_2, x_3, x_4, x_5\}$, $\mathcal{C}_3 = \{x_4, x_5, x_6\}$ et $\mathcal{C}_4 = \{x_3, x_7, x_8\}$, et la largeur d'arbre est de 3. Dans la suite, à partir d'une décompostion arborecente, nous considérons un arbre enraciné (I, F) où $\mathcal{C}_1$ est la racine, nous noterons $Fils(\mathcal{C}_i)$ l'ensemble des clusters fils de $\mathcal{C}_i$ et $Desc(\mathcal{C}_j)$ l'ensemble des variables qui appartiennent à $\mathcal{C}_j$ ou à un descendant $\mathcal{C}_k$ de $\mathcal{C}_j$ dans l'arbre enraciné en $\mathcal{C}_j$. Par exemple, $Desc(\mathcal{C}_2) = \mathcal{C}_2 \cup \mathcal{C}_3 = \{x_2, x_3, x_4, x_5, x_6\}$.

La première étape de BTD consiste à calculer une décomposition arborescente du graphe de contraintes. Cette décomposition arborescente permet d'obtenir un ordre partiel sur les variables permettant à BTD d'exploiter les propriétés structurelles du graphe et de couper dans l'arbre de recherche. En effet, les variables

verse, ApproxBTD commence directement avec un sous-problème triangulé pouvant être plus grand que le poly-arbre de la méthode précédente.

[3] Un graphe est triangulé si chaque cycle de longueur supérieur à quatre a une corde, *i.e* une arête reliant deux sommets non consécutifs dans le cycle.



(a) (b)

FIG. 1 – (a) Un graphe de contraintes sur 8 variables. (b) Une décomposition arborescente de ce graphe de contraintes.

sont affectées selon une recherche en profondeur d'abord à partir de la racine de la décomposition. Autrement dit, nous affectons les variables du cluster racine $\mathcal{C}_1$, puis celles de $\mathcal{C}_2$, celles de $\mathcal{C}_3$ etc... Par exemple, $x_1, x_2, \ldots, x_8$ est un ordre possible. De plus, la décomposition arborescente et l'ordre sur les variables permettent à BTD de diviser le problème $\mathcal{P}$ en plusieurs sous-problèmes. Etant donnés deux clusters $\mathcal{C}_i$ et $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$), le sous-problème enraciné en $\mathcal{C}_j$ dépend de l'affectation courante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$. On notera ce sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Son ensemble des variables est $Desc(\mathcal{C}_j)$. Le domaine de chaque variable appartenant à $\mathcal{C}_i \cap \mathcal{C}_j$ est réduit à sa valeur associée dans $\mathcal{A}$. Concernant l'ensemble des contraintes, il contient les contraintes qui impliquent au moins une variable apparaissant exclusivement dans $\mathcal{C}_j$ ou un de ses descendants. Considérons, par exemple, le CSP dont le graphe de contraintes est donné par la figure 1(a). Chaque domaine est $\{a, b, c, d\}$ et chaque contrainte $c_{ij} = \{x_i, x_j\}$ a une relation $r_{c_{ij}}$ telle que $x_i \neq x_j$. Soit $\mathcal{A} = (x_2 \leftarrow b, x_3 \leftarrow c)$, l'ensemble de variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ est $Desc(\mathcal{C}_2)$, (avec $d_{x_2} = \{b\}, d_{x_3} = \{c\}$ et $d_{x_4} = d_{x_5} = d_{x_6} = \{a, b, c, d\}$) et son ensemble de contraintes est $\{c_{24}, c_{25}, c_{34}, c_{35}, c_{45}, c_{46}, c_{56}\}$. Nous définissons la notion de *goods structurels*. Un good structurel de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$) est l'affectation courante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$ pouvant être étendue sur le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Par exemple, si nous considérons l'affectation $\mathcal{A} = (x_2 \leftarrow b, x_3 \leftarrow c)$ sur $\mathcal{C}_1 \cap \mathcal{C}_2$ nous obtenons le good $(\mathcal{A})$ grâce à l'affectation sur $Desc(\mathcal{C}_2)$ défini par $(x_2 \leftarrow b, x_3 \leftarrow c, x_4 \leftarrow a, x_5 \leftarrow d, x_6 \leftarrow b)$ satisfaisant toutes les contraintes de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$. Dans un premier temps cette affectation $(\mathcal{A})$ est explorée, puis lorsque son extension sur $\{x_4, x_5, x_6\}$ est valide, $(\mathcal{A})$ est enregistrée comme un good. Ainsi, si durant la recherche, l'affectation $\mathcal{A} = (x_2 \leftarrow b, x_3 \leftarrow c)$ est à nouveau étudiée, la recherche n'explore pas le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ car nous avons déjà prouvé qu'il existe une solution compatible avec $\mathcal{A}$. Au contraire, si aucune solution n'est trouvée pour une autre affectation $\mathcal{A}'$ sur $\mathcal{P}_{\mathcal{A}',\mathcal{C}_1/\mathcal{C}_2}$, un *nogood structurel* est alors enregistré. Ce nogood pourra être utilisé comme une nouvelle contrainte du problème.

Dans la section suivante, nous définirons la notion de good structurel pour le comptage, qui est basée sur les mêmes principes que les goods et nogoods structurels.

3 Dénombrement de solutions avec BTD

La méthode BTD est une adaptation de BTD pour le comptage de solutions. Comme pour BTD, la première étape de BTD consiste à calculer une décomposition arborescente du graphe de contraintes. Cette décompostion arborescente induit un ordre partiel sur les variables pour exploiter les propriétés structurelles du graphe et permettre d'élaguer l'arbre de recherche. Tandis que BTD utilise la notion de good, pour le dénombrement de solutions nous utilisons la notion de *#good*. Le but est de sauver le nombre de solutions des sous-problèmes induit par la décomposition arborescente. En effet, un # good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$) est une paire $(\mathcal{A}, nb)$ où $\mathcal{A}$ est une affectation $\mathcal{C}_i \cap \mathcal{C}_j$ et nb le nombre de solutions du sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Dans l'exemple de la Section 2, si nous considérons l'affectation $\mathcal{A} = (x_4 \leftarrow a, x_5 \leftarrow d)$ sur $\mathcal{C}_2 \cap \mathcal{C}_3$ nous obtenons un #good $(\mathcal{A}, 2)$ car nous avons deux solutions pour le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_2/\mathcal{C}_3}$.

BTD explore l'espace de recherche suivant l'ordre des variables induit par la décomposition arborescente. Ainsi, on commence par les

Algorithme 1 : BTD($\mathcal{A}$, $\mathcal{C}_i$, $V_{\mathcal{C}_i}$) : entier

```
if V_{C_i} = ∅ then
    if Fils(C_i) = ∅ then
        return 1
    else
        F ← Fils(C_i)
        NbSol ← 1
        while F ≠ ∅ et NbSol ≠ 0 do
            Choisir C_j in F
            F ← F − {C_j}
            if (A[C_i ∩ C_j], nb) n'est pas un #good dans P then
                nb ← BTD(A, C_j, V_{C_j} − (C_i ∩ C_j))
                enregistre le #good
                (A[C_i ∩ C_j], nb) de C_i/C_j dans P
            NbSol ← NbSol × nb ;
        return NbSol
else
    Choisir x ∈ V_{C_i}
    NbSol ← 0
    d ← d_x
    while d ≠ ∅ do
        Choisir v dans d
        d ← d-{x}
1       if A ∪ {x ← v} ne viole aucune c ∈ C then
            NbSol ← NbSol+BTD(A ∪ {x ← v},
            C_i, V_{C_i}-{x})
    return NbSol
```

variables du cluster racine $\mathcal{C}_1$. A l'intérieur du cluster $\mathcal{C}_i$, on affecte une valeur à une variable, on "backtrack" si une contrainte est violée. Ce schéma peut être amélioré avec le maintient de l'arc consistance. Lorsque toutes les variables de $\mathcal{C}_i$ sont affectées, BTD calcule le nombre de solutions du sous-problème induit par le premier fils de $\mathcal{C}_i$, s'il en existe un. Plus généralement, considérons $\mathcal{C}_j$ un fils de $\mathcal{C}_i$. Etant donnée une affectation courante $\mathcal{A}$ sur $\mathcal{C}_i$, BTD vérifie si l'affectation $\mathcal{A}[\mathcal{C}_i \cap C_j]$ correspond à un #good. Si c'est le cas, BTD multiplie le nombre de solutions enregistré avec le nombre de solutions de $\mathcal{C}_i$ avec $\mathcal{A}$ comme affectation. Sinon, on étend $\mathcal{A}$ sur $Desc(\mathcal{C}_i)$ dans l'ordre pour compter le nombre nb d'extensions consistantes et on enregistre le #good $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], nb)$. BTD calcule le nombre de solutions du sous-problème induit par le fils suivant de $\mathcal{C}_i$. Finalement lorsque chaque fils de $\mathcal{C}_i$ a été examiné, BTD essaye de modifier l'affectation courante de $\mathcal{C}_i$. Le nombre de solutions de $\mathcal{C}_i$ est la somme des solutions de chaque affectation.

BTD est décrit par l'algorithme 1. Etant donnée une affectation $\mathcal{A}$ et un cluster $\mathcal{C}_i$, BTD regarde le nombre d'extensions $\mathcal{B}$ de $\mathcal{A}$ sur $Desc(\mathcal{C}_i)$ tel que $\mathcal{A}[\mathcal{C}_i\text{-}V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i\text{-}V_{\mathcal{C}_i}]$. $V_{\mathcal{C}_i}$ est l'ensemble des variables non affectées de $\mathcal{C}_i$. Le premier appel est BTD$(\emptyset, \mathcal{C}_1, \mathcal{C}_1)$ et il retourne le nombre de solutions. Notons $\mathcal{S}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ le nombre de solutions d'un sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Ainsi, $\mathcal{S}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j} = \sum_{\mathcal{B}} \mathcal{S}_{\mathcal{B},\mathcal{C}_i/\mathcal{C}_j}$ pour chaque affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_j)$ tel que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$. Par exemple, avec $\mathcal{A} = (x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c)$ comme affectation de C_1 alors $\mathcal{S}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2} = \mathcal{S}_{\mathcal{A}\cup(x_4\leftarrow a,x_5\leftarrow d),\mathcal{C}_1/\mathcal{C}_2} + \mathcal{S}_{\mathcal{A}\cup(x_4\leftarrow d,x_5\leftarrow a),\mathcal{C}_1/\mathcal{C}_2} = 2 + 2 = 4$. Soit $\mathcal{V}ar(\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j})$ l'ensemble des variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ moins les variables de $\mathcal{C}_i \cap \mathcal{C}_j$. Autrement dit, c'est l'ensemble des variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ non affectées. Si $\mathcal{C}_i$ a k fils : $\mathcal{C}_{i_1}, ..., \mathcal{C}_{i_k}$ alors $\cap_{1\leqslant j\leqslant k}\mathcal{V}ar(\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_{i_j}}) = \emptyset$. On note $\mathcal{S}_{\mathcal{A},\mathcal{C}_i}$ le nombre de solutions de $Desc(\mathcal{C}_i)$ avec l'affectation $\mathcal{A}$ sur $\mathcal{C}_i$. D'où, $\mathcal{S}_{\mathcal{A},\mathcal{C}_i} = \prod_{1\leqslant j\leqslant k} \mathcal{S}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_{i_j}}$. Par exemple, le nombre de solutions de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ (avec l'affectation $\mathcal{A} = (x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c)$ est 4 sur (x_4, x_5, x_6) et 6 solutions pour $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_4}$ sur (x_7, x_8). Donc, il y a 24 solutions sur $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ avec $\mathcal{A}$. Notons que pour $[\mathcal{C}_1 \cap \mathcal{C}_4]$, $((x_3 \leftarrow c), 6)$ est un #good. Pour une autre affectation sur $\mathcal{C}_1$, e.g $\mathcal{A}' = (x_1 \leftarrow b, x_2 \leftarrow a, x_3 \leftarrow c)$, il n'est pas nécessaire de calculer les solutions sur $\mathcal{C}_4$ car le #good $((x_3 \leftarrow c), 6)$ peut être utilisé pour $\mathcal{A}'$.

La complexité en espace de BTD est $O(n.s.d^s)$ et celle en temps est $O(n.m.d^{w+1})$ avec $w+1$ la taille du plus grand cluster $\mathcal{C}_k$ et s la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ ($\mathcal{C}_j$ un fils de $\mathcal{C}_i$).

En pratique, pour les problèmes ayant une grande largeur d'arbre, BTD explose en temps et en mémoire, voir à la Section 5. Dans ce cas, nous sommes intéressés par une méthode d'approximation.

4 Approximation avec BTD

Nous considérons ici des CSPs qui ne sont pas nécessairement structurés. Pour les résoudre, nous proposons d'exploiter BTD en définissant une nouvelle méthode d'approximation appelée ApproxBTD.

Sans perte de généralité, nous considérons le

cas de CSP binaires dans la présentation de la méthode. Nous pouvons définir une collection de sous-problèmes en partitionnant l'ensemble des contraintes, c'est à dire dans le cas de CSP binaires, en partitionnant l'ensemble des arêtes du graphe de contraintes. Nous remarquons que chaque graphe (X, C) peut être partitionné en k sous-graphes (X_1, E_1), ... , (X_k, E_k) tels que $\cup X_i = X$, $\cup E_i = C$ et $\cap E_i = \emptyset$ et tel que chaque sous-graphe (X_i, E_i) soit triangulé. Ainsi, chaque (X_i, E_i) peut être associé à un sous-problème structuré $\mathcal{P}_i$ (avec l'ensemble de variables X_i et l'ensemble de contraintes correspondant à E_i), qui peut être résolu efficacement en utilisant BTD. Ce partitionnement s'appuie sur le fait qu'il est facile de trouver une décomposition arborescente de largeur d'arbre minimum pour un graphe triangulé (*théorème de Fulkerson et Gross, 1965*). Supposons que $\mathcal{S}_{\mathcal{P}_i}$ est le nombre de solutions pour chaque sous-problème $\mathcal{P}_i$, $1 \leqslant i \leqslant k$. Nous estimerons le nombre de solutions de $\mathcal{P}$ en exploitant la propriété suivante. Premièrement, nous dénotons $\mathbb{P}_{\mathcal{P}}(\mathcal{A})$la probabilité de $\mathcal{A}$ *est une solution de* $\mathcal{P}$. $\mathbb{P}_{\mathcal{P}}(\mathcal{A}) = \dfrac{\mathcal{S}_{\mathcal{P}}}{\prod_{x \in X} d_x}$.

Propriété 1. *Soit un CSP donné* $\mathcal{P} = (X, D, C, R)$ *et une partition* $\{\mathcal{P}_1, ..., \mathcal{P}_k\}$ *de* $\mathcal{P}$ *induite par une partition de* C *en* k *éléments.*

$$\mathcal{S}_{\mathcal{P}} \approx \left\lceil \left(\prod_{i=1}^{k} \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \right) \times \prod_{x \in X} d_x \right\rceil$$

Notons que l'approximation retourne une réponse exacte si tous les sous-problèmes sont indépendants ($\cap X_i = \emptyset$) ou $k = 1$ ($\mathcal{P}$ est déjà triangulé) ou si il existe un sous-problème incohérent $\mathcal{P}_i$. De plus, nous pouvons fournir un majorant trivial du nombre de solutions dû au fait que chaque sous-problème $\mathcal{P}_i$ est une relaxation de $\mathcal{P}$ (le même argument est utilisé dans [21] pour construire un majorant).

$$\mathcal{S}_{\mathcal{P}} \leq \min_{i \in [1,k]} \left\lceil \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \times \prod_{x \in X} d_x \right\rceil$$

Notre méthode appelée ApproxBTD est décrite par l'algorithme 2. Appliqué sur un problème $\mathcal{P}$ avec un graphe de contraintes (X, C),

Algorithme 2 : ApproxBTD($\mathcal{P}$) : integer

```
Soit G' = (X', C') un graphe de contraintes associé à P ;
i ← 0 ;
while G' ≠ ∅ do
    i ← i + 1 ;
    Calculer un sous-graphe partiel triangulé (X_i, E_i) de G' ;
    Soit P_i le sous-problème associé à (X_i, E_i) ;
    S_{P_i} ←BTD(∅, C'_1, C'_1) avec C'_1 le cluster racine de la décomposition arborescente de P_i ;
    G' ← (X', C' − E_i) avec X' l'ensemble des variables induites par C'-E_i ;
k ← i ;
return ⌈ ∏_{i=1}^{k} S_{P_i} / ∏_{x∈X_i} d_x × ∏_{x∈X} d_x ⌉ ;
```

la méthode construit une partition $\{E_1, ..., E_k\}$ de C telle que le graphe de contraintes (X_i, E_i) est triangulé pour tout $1 \leqslant i \leqslant k$. Les sous-problèmes associés aux (X_i, E_i) sont résolus par BTD. La méthode retourne une approximation du nombre de solutions de $\mathcal{P}$ en utilisant la propriété 1.

Le nombre d'itérations de ApproxBTD est borné par n (nous avons au moins $n-1$ arêtes (un arbre) à chaque itération ou bien des sommets ont été effacés). Chaque sous-graphe triangulé et sa décomposition arborescente optimale associée peuvent être calculé en $O(nm)$ opérations [7][4]. De plus, nous garantissons que la largeur d'arbre w (plus un) de chaque sous-graphe triangulé produit est au plus égale à K, la taille de la plus grande clique (*clique number*) de $\mathcal{P}$. Soit w^* la largeur d'arbre (minimale) de $\mathcal{P}$, nous avons $w + 1 \leq K \leq w^* + 1$. Finalement, la complexité temporelle de ApproxBTD est $O(n^2 m d^K)$ et sa complexité mémoire est $O(nKd^{K-1})$.

5 Résultats expérimentaux

Nous avons effectué nos expérimentations sur des CSP aléatoires et sur des benchmarks SAT. Les expérimentations ont été réalisées sur

[4]Notons que cette approche retourne un sous-graphe maximal au sens de l'inclusion en nombre de contraintes pour un CSP binaire seulement. Dans le cas d'un CSP non-binaire (également en SAT), nous ne garantissons pas la maximalité du sous-graphe et ajoutant dans le sous-problème toutes les contraintes / clauses dont la clique associée d'arêtes est totalement incluse dans le sous-graphe.

un Pentium IV 3.2 GHz (resp. Xeon 2, 66 GHz) avec 1 Go (resp. 32 Go) pour les instances CSP (resp. SAT). Nous avons limité à une heure le temps autorisé pour résoudre chaque instance. Dans BTD à la ligne 1, nous utilisons Forward Checking au lieu de Backward Checking pour des raisons d'efficacité. A l'intérieur des clusters l'ordre dynamique *min domaine / max degré* est utilisé pour le choix de variables.

5.1 Instances aléatoires de CSP structurés

Nous étudions et comparons ApproxBTD avec BTD sur des instances structurées générées aléatoirement. Pour cela, nous considérons des instances aléatoires de k-arbres partiels avec les paramètres $(n, d, r_{max}, t, s_{max}, nc, nr)$ où chaque graphe de contraintes est un arbre de nc cliques et nr est le pourcentage d'arêtes supprimées dans les cliques. Il y a n variables avec un domaine de taille d, la taille de la plus grande clique est r_{max}, et la taille de la plus grande intersection est s_{max}. Chaque contrainte a une dureté égale à t, le pourcentage de tuples interdits parmi les tuples possibles pour la contrainte.

Les résultats présentés dans les Figures 2 et 3 sont les résultats du nombre de solutions et du temps CPU en millisecondes pour les classes $(50, 15, 8, t, 3, 10, 30\%)$ avec la dureté t variant entre 40% et 65%.

Chaque point dans la Figure 2 représente l'évaluation trouvée par ApproxBTD par rapport au nombre de solutions trouvé par BTD pour chaque instance. Nous observons que plus il y a de solutions, meilleure est l'approximation.

Le temps requis pour ces approximations est illustré par la Figure 3. Lorsque la dureté est inférieur à 53%, ApproxBTD est plus rapide que BTD. Pour les plus grandes duretés BTD peut être plus rapide que ApproxBTD car il résoud seulement un problème contre une collection de sous-problèmes.

5.2 Benchmarks SAT

Les benchmarks SAT viennent de `www.satlib.org`. Nous avons sélectionné les instances satisfaisables académiques (Tours de

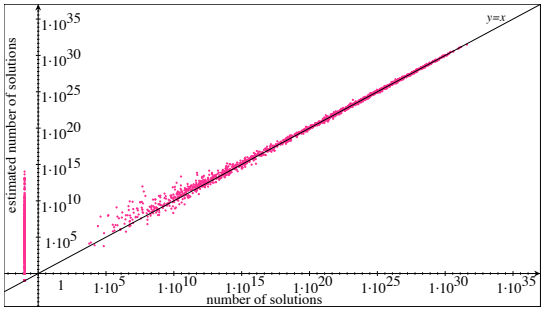


FIG. 2 – Comparaison entre le nombre de solutions trouvé par BTD et l'estimation trouvée par ApproxBTD sur les CSP structurés aléatoires (les points d'abscisses -0.1 représentent les instances inconsistantes).

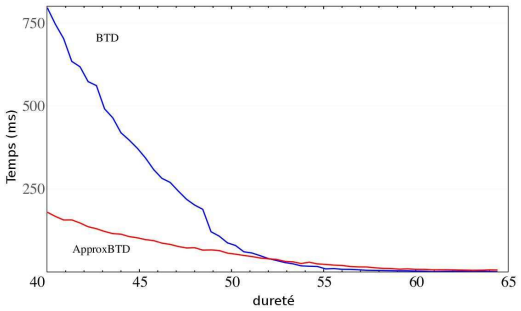


FIG. 3 – Temps CPU en millisecondes pour BTD et ApproxBTD sur les CSP structurés aléatoires.

Hanoi *hanoi*, All-Interval Series *ais*) et industrielles (analyse de faute de circuit *ssa* et *2bit*). Nous comparons BTD avec le compilateur d-DNNF `c2d` [6] qui exploite également la structure du problème. Ces deux méthodes utilisent l'ordre d'élimination de variables *MinFill* (sauf pour *hanoi* où nous avons utilisé l'ordre par défaut) pour construire une décomposition arborescente / d-DNNF. Nous avons également comparé ApproxBTD avec une méthode d'approximation nommée `SampleCount` [12] qui donne une estimation du minorant du nombre de solutions avec un intervalle de confiance élevé. La Table 1 résume nos résultats. Les colonnes sont le nom de l'instance, le nombre de

Instances	Vars	Clauses	w	Solutions	c2d	BTD	ApproxBTD			SampleCount	
					Temps	Temps	w	Solutions	Temps	Solutions	Temps
académiques											
ais6	61	581	41	24	0.04	0.06	5	≈ 1	0.09	≥ 6	0.184
ais8	113	1520	77	44	0.51	2.34	7	≈ 1	0.52	≥ 8	41.31
ais10	181	3151	116	296	17.14	390.32	9	≈ 1	2.69	≥ 38	384.8
ais12	265	4269	181	1328	1162.75	-	11	≈ 1	2.69	≥ 0	17.6
hanoi4	718	4934	46	1	3.41	1.72	6	≈ 1	1.57	≥ 0	5.26
hanoi5	1931	14468	58	1	-	25.46	7	≈ 1	14.98	≥ 0	6.19
industrielles											
ssa7552-038	1501	3575	25	2.84e40	0.15	0.72	7	$\approx$ 6.34e35	1.36	$\geq$ 6.67e38	1763
ssa7552-158	1363	3034	9	2.56e31	0.10	0.19	5	$\approx$ 2.59e27	0.79	$\geq$ 3.57e29	175
ssa7552-159	1363	3032	11	7.66e33	0.09	0.25	5	$\approx$ 1.09e30	0.84	$\geq$ 1.62e32	237
ssa7552-160	1391	3126	12	7.47e32	0.12	0.29	5	$\approx$ 2.88e33	0.99	$\geq$ 2.02e31	1117
2bitcomp_5	125	310	36	9.84e15	0.47	11.53	6	$\approx$ 2.23e15	0.02	$\geq$ 1.80e15	1.42
2bitmax_6	252	766	58	2.10e29	18.71	-	7	$\approx$ 1.98e28	0.1	$\geq$ 1.02e28	8.42

TAB. 1 – Solution counting for SAT instances. Time in seconds. A "-" means the instance was not solved in less than 1 hour.

variables booléennes, le nombre de clauses, la largeur d'arbre de la décomposition arborescente, le nombre exact de solutions, le temps CPU en secondes pour `c2d` et BTD, pour ApproxBTD : la largeur d'arbre maximale pour tous les sous-problèmes triangulés, le nombre de solutions approché et le temps, enfin pour `SampleCount` : le minorant du nombre de solutions et le temps. Nous remarquons que BTD peut résoudre des instances ayant une petite largeur d'arbre. `c2d` est généralement plus rapide (excepté pour *hanoi5*) mais il souffre également lorsqu'on a une grande largeur d'arbre (e.g. *ais*). Notre méthode d'approximation ApproxBTD exploite une partition du graphe de contraintes tels que les sous-problèmes résultants aient une petite largeur d'arbre ($w \leq 11$) comme le montrent les résultats. En pratique la méthode permet d'obtenir des résultats relativement rapidement même si la largeur d'arbre originale est importante[5]. La qualité de l'appoximation trouvée par ApproxBTD est relativement bonne et est comparable à `Samplecount` qui prend plus de temps et requiert un réglage de paramètres (nous utilisons les paramètres avec $t = 7, s = 20, \alpha = 1$ les options par défaut). Le majorant calculé par ApproxBTD n'est pas mentionné car il est très supérieur au nombre de solutions sur ces instances.

[5]Une petite largeur d'arbre ne permet pas en pratique d'être rapide à chaque fois (voir e.g. les instances ssa)

6 Discussion and conclusion

Dans cet article, nous avons proposé deux méthodes pour résoudre le problème de comptage du nombre de solutions d'un CSP. Ces méthodes exploitent une décomposition arborescente des CSPs. Nous avons présenté une méthode exacte qui est adaptée aux problèmes ayant une petite largeur d'arbre. Pour des problèmes avec une largeur d'arbre importante et un graphe de contraintes peu dense, nous avons présenté une nouvelle méthode d'approximation dont la qualité des résultats obtenus est proche des méthodes existantes mais qui sont obtenus bien plus rapidement et sans avoir à effectuer des réglages complexes de paramètres, à l'exception du choix de l'heuristique de décomposition arborescente.

Remerciements

Ce travail est partiellement supporté par le projet ANR STAL-DEC-OPT.

Références

[1] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings

in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[2] R. Bayardo and J. Pehoushek. Counting models using connected components. In *AAAI-00*, pages 157–162, 2000.

[3] R. Burton and J. Steif. Nonuniqueness of measures of maximal entropy for subshifts of finite type. In *Ergodic theory and dynamical system*, 1994.

[4] A. Choi and A. Darwiche. An edge deletion semantics for belief propagation and its practical impact on approximation quality. In *Proc. of AAAI*, pages 1107–1114, 2006.

[5] A. Darwiche. On the tractable counting of theory models and its applications to truth maintenance and belief revision. *Journal of Applied Non-classical Logic*, 11 :11–34, 2001.

[6] A. Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332, 2004.

[7] P.M Dearing, D.R. Shier, and D.D. Warner. Maximal chordal subgraphs. *Discrete Applied Mathematics*, 20(3) :181–190, 1988.

[8] R. Dechter and R. Mateescu. The impact of and/or search spaces on constraint satisfaction and counting. In *Proc. of CP*, pages 731–736, Toronto, CA, 2004.

[9] R. Dechter and R. Mateescu. And/or search spaces for graphical models. *Artif. Intell.*, 171(2-3) :73–106, 2007.

[10] V. Gogate and R. Dechter. Approximate counting by sampling the backtrack-free search space. In *Proc. of AAAI-07*, pages 198–203, Vancouver, CA, 2007.

[11] V. Gogate and R. Dechter. Approximate solution sampling (and counting) on and/or search spaces. In *Proc. of CP-08*, pages 534–538, Sydney, AU, 2008.

[12] C. P. Gomes, J. Hoffmann ans A. Sabharwal, and B. Selman. From sampling to model counting. In *Proc. of IJCAI*, pages 2293–2299, 2007.

[13] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting : A new strategy for obtaining good bounds. In *Proc. of AAAI*, 2006.

[14] C P. Gomes, W-J. van Hoeve, A. Sabharwal, and B. Selman. Counting CSP solutions using generalized XOR constraints. In *Proc. of AAAI-07*, pages 204–209, Vancouver, BC, 2007.

[15] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[16] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[17] K. Kask, R. Dechter, and V. Gogate. New look-ahead schemes for constraint satisfaction. In *Proc. of AI&M*, 2004.

[18] L. Kroc, A. Sabharwal, and B. Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *Proc. of CPAIOR-08*, pages 127–141, Paris, France, 2008.

[19] T.K Satish Kumar. A model counting characterization of diagnoses. In *Proc. of the 13th International Workshop on Principles of Diagnosis*, 2002.

[20] M. Mann, G. Tack, and S. Will. Decomposition during search for propagation-based constraint. In *CoRR abs/0712.2389 :*, 2007.

[21] G. Pesant. Counting solutions of CSPs : A structural approach. In *Proc. of IJCAI*, pages 260–265, 2005.

[22] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[23] D. Roth. On the hardness of approximate reasonning. *Artificial Intelligence*, 82(1-2) :273–302, 1996.

[24] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, number

874 in LNCS, Rosario, Orcas Island (WA), May 1994. Springer-Verlag.

[25] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *SAT-04*, Vancouver, Canada, 2004.

[26] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Sciences*, 8 :189–201, 1979.

[27] W. Wei and B. Selman. A new approach to model counting. In *Proc. of SAT-05*, pages 324–339, St. Andrews, UK, 2006.